cctbx news

Ralf W. Grosse-Kunstleve[a], Peter H. Zwart[a], Pavel V. Afonine[a], Thomas R. Ioerger[b], Paul D. Adams[a]

[a]Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA and [b]Texas A&M University, College Station, TX 77843, USA. E-mail: rwgk@cci.lbl.gov ; WWW: http://cci.lbl.gov/

Abstract

We describe recent developments of the Computational Crystallography Toolbox.

Preamble

In order to interactively run the examples scripts shown below, the reader is highly encouraged to visit http://cci.lbl.gov/cctbx_build/ and to download one of the completely self-contained, self-extracting binary cctbx distributions (supported platforms include Linux, Mac OS X, Windows, IRIX, and Tru64 Unix). All example scripts shown below were tested with cctbx build 2006_11_22_0037.

In the following we refer to our articles in the previous editions of this newsletter as "Newsletter No. 1", "Newsletter No. 2", etc. to improve readability. The full citations are included in the reference section.

1 Introduction

The *Computational Crystallography Toolbox* (cctbx, http://cctbx.sourceforge.net/) is the open-source component of the Phenix project (http://www.phenix-online.org/). Most recent cctbx developments are geared towards supporting new features of the `phenix.refine` application. Thus, the open-source `mmtbx` (macromolecular toolbox) module is currently being most rapidly developed. In this article we give an overview of some of the recent developments. However, the main theme of this article is the presentation of a light-weight example command-line application that was specifically developed for this newsletter: sequence alignment and superposition of two molecules read from files in PDB format. This involves parameter input based on the Phil module presented in Newsletter No. 5, fast reading of the PDB files with the new `iotbx.pdb.input` class, simple sequence alignment using the new `mmtbx.alignment` module, and use of the Kearsley (1989) superposition algorithm to find the least-squares solution for superposing C-alpha positions. The major steps are introduced individually, followed by a presentation of the complete application.

The example application is deliberately limited in functionality to make it concise enough for this article. The main goal is to show how the open-source components are typically combined into an application. Even though the example is quite specific to macromolecular crystallography, we believe it will also be useful for a small-molecule audience interested in utilizing the large open-source library of general crystallographic algorithms (see our previous articles in this newsletter series) to build an application.

2 iotbx.pdb

The PDB format is the predominant working format for atomic parameters (coordinates, occupancies, displacement parameters, etc.) in macromolecular crystallography, but many small-molecule programs also support this format. `phenix.refine` also utilized the PDB format, mainly to facilitate easy communication with other programs, most notably graphics programs for visualization.

2.1 iotbx.pdb.input

The PDB format specifications are available at http://www.pdb.org/ . Technically, the format is very simple, therefore a vast number of parsers exist in scientific packages. The cctbx is no exception. A parser implemented in Python has been available for several years. In many cases Python's runtime performance is sufficient for interactive processing of PDB files, but can be limiting for large files, or for traversing the entire PDB database (currently 40731 files, about 25 GB total). This has prompted us to implement a fast parser in C++, complete with Python bindings in the same style as all other cctbx C++ classes, comprehensive error reporting, and fully automatic memory life-time management (no manual new/delete or malloc/free). Reading a PDB file from Python is simple:

```
import iotbx.pdb
pdb_inp = iotbx.pdb.input(file_name="pdb1htq.ent")
```

With a size of 76 MB this is the largest file in the PDB, but full processing takes only 3.8 s on a 2.6 GHz Opteron, of which about 0.5 s are for simply transferring the data from disk into memory. In contrast, the older Python implementation needs 89 s for processing the file into data structures of similar complexity. In the future all our cctbx-based applications will make use of the new, faster parser. Interestingly, when processing the PDB database with `iotbx.pdb.input` using multiple CPUs, disk-I/O is the rate limiting step. Using 8 CPUs, we can process all 25 GB in less than five minutes. Using more CPUs does not reduce this time.

The `pdb_inp` object holds the information from the PDB file in a structured way. The "sections" of the file according to the PDB format specifications at http://www.pdb.org/ are available as, e.g.:

```
pdb_inp.title_section()
pdb_inp.remark_section()
pdb_inp.crystallographic_section()
...
```

In Python these sections appear as simple lists of strings. The full power of Python and the cctbx libraries is available for post-processing this information. Since most sections are never very large, there is no point in writing specialized C++ processing code, which is typically significantly more labor intensive compared to writing equivalent Python code, and much more difficult to adjust for new developments.

The only section of PDB files that is sometimes found to be very large is the "coordinate section" with the ATOM and HETATM records. This section is fully processed in the `iotbx.pdb.input` constructor shown above. The corresponding information is available via the methods:

```
labels_list = pdb_inp.input_atom_labels_list()
atoms = pdb_inp.atoms()
```

which return arrays of the same length, each with one data object per atom. The information for one atom is again accessible from Python, e.g.:

```
for labels,atom in zip(labels_list, atoms):
  print labels.chain(), labels.resname(), labels.name(), atom.xyz, atom.b
```

2.2 iotbx.pdb.hierarchy

The `input_atom_labels` objects in the `pdb_inp.input_atom_labels_list()` above store the `name`, `resname`, `chain`, `icode` (insertion code), `segid` (segment identifier), and `altloc` (alternate location indicator) for each atom. This information defines a hierarchical organization of the macromolecules, but in a highly redundant way which complicates further processing steps, such as the assignment of geometry restraints (see Newsletter No. 4). `iotbx.pdb.input` supports building an `iotbx.pdb.hierarchy` object. The redundant atom labels are analyzed to build a non-redundant six-deep `hierarchy` object, e.g.:

```
hierarchy = pdb_inp.construct_hierarchy()
```

The six-deep data structure consists of:

```
hierarchy
  model
    chain
      conformer
        residue
          atom
```

which can be concisely traversed from Python:

```
for model in hierarchy.models():
  for chain in model.chains():
    for conformer in chain.conformers():
      for residue in conformer.residues():
        for atom in residue.atoms():
          # ...
```

The time for building the `hierarchy` object given `pdb1htq.ent` is about 0.7 s. Traversing the hierarchy with the five-deep loop above takes only about 0.6 s, i.e. is unlikely to be a rate-limiting step even for very large structures. Therefore the few lines of example code given in this section are probably one of the most convenient and efficent ways of quickly processing a PDB file from a scripting language.

For general information on how to learn more about Python objects, look under the "Tutorials Siena 2005" link at cctbx.sf.net. For example, the command:

```
libtbx.help iotbx.pdb.residue
```

will show the complete interface of the `residue` object.

`hierarchy` objects can be manipulated or constructed from scratch from both Python and C++. However, high-level functionality like inserting or deleting residues or chains, or formatting output is currently not available. We will add such high-level manipulations as the need arises. The typical development process is to implement a required high-level operation given the currently available interfaces, then add it as a new method to the most suitable existing class to make it easily accessible for other purposes. We expect the hierarchy objects to continuously grow in this way for some time to come.

2.3 pdb_inp.xray_structure_simple

The `xray_structure_simple` simple method of `iotbx.pdb.input` is an efficient implementation converting the information stored in the `pdb_inp` object above to a list of `cctbx.xray.scatterer` objects, managed by the `cctbx.xray.structure` class. See our previous newsletter articles and the Siena 2005 tutorials for various examples on how to work with these objects.

Fundamentally, the conversion is trivial. Each input atom is converted to exactly one `xray.scatterer`. However, as always, the devil is in the details. The PDB `CRYST1` and `SCALE` cards have to be evaluated to obtain the correct fractionalization matrix (PDB coordinates are with respect to a Cartesian basis). The trickest problem is the determination of the scattering type for each atom, for which three PDB columns have to be considered (atom name, element symbol, charge). Following the PDB format specifications strictly, the scattering type is clearly defined, but unfortunately deviations from the strict specifications are quite common. For example, the element symbol may be missing or mis-aligned, or the charge symbol is sometimes found to be given as "+2" instead of "2+". The `xray_structure_simple` method allows for some deviations from the strict PDB specifications as long as the error is highly obvious. More serious errors are communicated via exceptions with carefully formatted, informative error messages.

The `xray_structure_simple` method is relatively expensive in terms of runtime, partially because the site symmetry is determined for each atom, which involves looping over the symmetry elements and distance calculations. The runtime for the `pdb1htq.ent` structure (978720 atoms) is about 9.9 s. However, this step is typically performed only once at the start of a program. To put this further into context, a structure factor calculation up to a resolution of 3 A, using the FFT method, takes about 26 s. This underlines that the time for I/O using the new `pdb.input` class is generally neglible in the context of a whole application.

For completeness, the code for building the `xray.structure` and computing the structure factors is:

```
xray_structure = pdb_inp.xray_structure_simple()
xray_structure.structure_factors(d_min=3, algorithm="fft")
```

3 mmtbx.alignment

`mmtbx.alignment` provides algorithms for aligning two protein sequences, where each sequence is represented as a string of one-letter amino-acid codes. The implementation is based on the ideas of Gotoh (1982) and runs in quadratic time O(M*N), where M and N are the sequence lengths. It does both global (Needleman & Wunsch, 1970) and local (Smith & Waterman, 1981) alignments, assuming affine (linear) gap penalties (for which default gap-cost parameters may be changed by the user). Alignments are based on maximizing similarity. Similarity scores between amino acids are specified via symmetric matrices. Similarity matrices of Dayhoff (1978) and BLOSUM50 (Henikoff & Henikoff (1992), http://en.wikipedia.org/wiki/BLOSUM) are provided. User-supplied matrices are also supported (this feature also enables alignment of non-amino-acid sequences).

To show a short example of aligning two sequences:

```
from mmtbx.alignment import align
align_obj = align(
  seq_a="AESSADKFKRQHMDTEGPSKSSPTYCNQMM",
  seq_b="DNSRYTHFLTQHYDAKPQGRDDRYCESIMR")
```

The `align_obj` holds matrices used in the dynamic-programming (http://en.wikipedia.org/wiki/Dynamic_programming) alignment algorithm. Methods are available to get the alignment score and to extract the actual sequence alignment:

```
print "score: %.1f" % align_obj.score()

alignment = align_obj.extract_alignment()
print alignment.match_codes
print alignment.a
print alignment.matches(is_similar_threshold=0)
print alignment.b
```

The output is:

```
score: 6.0
mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
AESSADKFKRQHMDTEGPSKSSPTYCNQMM
  |    |  || |              |
DNSRYTHFLTQHYDAKPQGRDDRYCESIMR
```

`match_codes` is a string of the characters `m`, `i`, and `d`, for match, insertion, and deletion, respectively. The alignment above is based on identity matches only. Alternatively, the similarity matrices of `"dayhoff"` or `"blosum50"` can be used, e.g.:

```
align_obj = align(
  seq_a="AESSADKFKRQHMDTEGPSKSSPTYCNQMM",
  seq_b="DNSRYTHFLTQHYDAKPQGRDDRYCESIMR",
```

```
  style="global",
  gap_opening_penalty=10,
  gap_extension_penalty=2,
  similarity_function="blosum50")
```

The output of the same print statements as above is in this case:

```
mmmmmmmmmmmmmmmmmdmmmmmmmmmmmmmmi
AESSADKFKRQHMDTEGPSKSSPTYCNQMM-
  |    |  || | * |      ||  *|
DNSRYTHFLTQHYDAK-PQGRDDRYCESIMR
```

Here the `*` indicate residues that are similar above the `is_similar_threshold`, using the `blosum50` similarity matrix.

Alignment of sequences of typical lengths is fast enough for interactive work (300 residues: about 1 s), but can take minutes given thousands of residues (2200 residues: about 40 s). Also, for very long sequences the memory consumption can be significant since four M*N alignment matrices are stored as pure Python objects. In the future we may reimplement the core of the alignment algorithm in C++ to increase runtime performance (by a factor 30-50) and to significantly reduce memory consumption. However, the Python interface presented here is expected to stay the same.

4 scitbx.math.superpose

The `scitbx.math.superpose` module implements the quaternion method of Kearsley (1989) for superpositioning two related vector sets. In comparison to the related method of Kabsch (1976), this method has the advantage of gracefully handling degenerate situations (e.g. if all atoms are on a straight line) without the need for handling special cases (Kabsch, 1978). For all non-degenerate situations, the results of the Kabsch and Kearsley methods are identical within floating point precision.

We will give a self-contained example, using the `iotbx.pdb.input` class discussed above. First, we define a few atoms to be aligned:

```
  # residues in PDB entry 1AON
  gly1 = """\
ATOM     55  N   GLY A   9      47.072 -70.250  -4.389  1.00 27.28
ATOM     56  CA  GLY A   9      45.971 -69.823  -3.545  1.00 22.53
ATOM     57  C   GLY A   9      45.946 -68.397  -3.056  1.00 25.45
ATOM     58  O   GLY A   9      46.350 -67.467  -3.764  1.00 28.17
""".splitlines()
  gly2 = """\
ATOM    134  N   GLY A  19      46.795 -55.602  -6.961  1.00 15.66
ATOM    135  CA  GLY A  19      47.081 -55.164  -8.320  1.00 11.86
ATOM    136  C   GLY A  19      45.844 -54.551  -8.936  1.00  7.75
ATOM    137  O   GLY A  19      45.851 -53.398  -9.384  1.00 10.45
""".splitlines()
```

This code produces two Python lists of Python strings. To be compatible with the `iotbx.pdb.input` constructor, these have to be converted to C++ arrays ("flex arrays", see Newsletter No. 1):

```
from cctbx.array_family import flex
gly1 = flex.std_string(gly1)
gly2 = flex.std_string(gly2)
```

Now we are ready to instantiate two `iotbx.pdb.input` objects. Instead of reading directly from a file as shown before, we read the PDB `lines` from the C++ array of strings:

```
import iotbx.pdb
pdb1 = iotbx.pdb.input(source_info=None, lines=gly1)
pdb2 = iotbx.pdb.input(source_info=None, lines=gly2)
```

Next, we extract two C++ flex arrays with the coordinates:

```
sites1 = pdb1.extract_atom_xyz()
sites2 = pdb2.extract_atom_xyz()
```

These arrays can be used directly to instantiate the `least_squares_fit` class which computes the rotation and translation for the best fit using the algorithm of Kearsley (1989) (which is the default):

```
from scitbx.math import superpose
superposition = superpose.least_squares_fit(
  reference_sites=sites1,
  other_sites=sites2)
```

The `r` and `t` attributes are `scitbx.matrix` instances which provide `mathematica_form()` methods which we use here for pretty-printing:

```
print superposition.r.mathematica_form(
  label="r", one_row_per_line=True, format="%8.5f")
print superposition.t.mathematica_form(
  label="t", format="%8.5f")
```

The output is:

```
r={{-0.72436, -0.03369,  0.68860},
   {-0.43741,  0.79448, -0.42127},
   {-0.53289, -0.60635, -0.59023}}
t={{83.88229}, {-8.78874}, {-17.07881}}
```

To compute the RMS difference of the superposed sites:

```
sites2_fit = superposition.other_sites_best_fit()
print "rms difference: %.4f" % sites1.rms_difference(sites2_fit)
```

Output:

```
rms difference: 0.3671
```

The following code produces a listing of distances for each atom:

```
from scitbx import matrix
lbls1 = pdb1.input_atom_labels_list()
lbls2 = pdb2.input_atom_labels_list()
for l1,s1,l2,s2f,d in zip(lbls1, sites1,
                          lbls2, sites2_fit,
                          sites2_fit-sites1):
  print l1.pdb_format(), "%8.4f, %8.4f, %8.4f" % s1
  print l2.pdb_format(), "%8.4f, %8.4f, %8.4f" % s2f
  print "      difference: %8.4f, %8.4f, %8.4f" % d, \
        "|d| = %6.4f" % abs(matrix.col(d))
  print
```

The input atom labels are obtained from the `iotbx.pdb.input` objects using the `pdb_format()` method which returns the atom labels in the same arrangement as found in the PDB file; this output is useful for locating an atom in the original file. `sites2_fit-sites1` uses flex array algebra to compute the difference vectors, and the `scitbx.matrix` class is used to compute the length of the vector. The output starts with:

```
" N   GLY A   9 "  47.0720, -70.2500,  -4.3890
" N   GLY A  19 "  47.0655, -70.5000,  -4.1925
     difference:  -0.0065,  -0.2500,   0.1965 |d| = 0.3180
...
```

It is easy to verify that the Kabsch method gives identical results for this non-degenerate configuration of atoms:

```
superposition_kabsch = superpose.least_squares_fit(
  reference_sites=sites1,
  other_sites=sites2,
  method="kabsch")
```

In this case the rotation matrix and the translation vector are exactly identical to the Kearsley results shown above. However, `method="kabsch"` should not be used in applications since we didn't spent the effort of writing code for special cases. Therefore the Kearsly method is the default.

5 Putting the pieces together: mmtbx.super

The large variety of tools in the `cctbx` enables quick develop of small scripts (a.k.a. jiffies) that perform non-trivial tasks with relative ease.

In a typical jiffy, most work goes into building a user interface that facilitates the communication between program and user. The actual computational work that needs to be done is often not very laborious, and can be as trivial as a simple call of a library function.

The following paragraphs will illustrate the rapid development of a lightweight structure superposition command line tool using a variety of recent additions to the cctbx. The full code can be found in `cctbx_sources/mmtbx/mmtbx/command_line/super.py`. This script is also available from the command line under the name `mmtbx.super`.

5.1 Design

The goal is to develop a simple tool that carries out the following tasks:

1. Determine input file names and alignment parameters (user interface).
2. Read in two related PDB files.
3. Determine corresponding residues between the two PDB files.
4. Compute a least-squares superposition of C-alpha atoms.
5. Write out the superposed coordinates to a new PDB file.

5.2 User interface

A basic, yet versatile, user interface can be implemented in a very straightforward manner using Phil. Since Phil has been discussed at length in Newsletter No. 5, we limit ourselves to a brief overview of the implementation.

The Phil "master parameters" definition embedded in `super.py` is:

```
import libtbx.phil
master_params = libtbx.phil.parse("""\
super {
  fixed = None
    .type = str
  moving = None
    .type = str
  moved = "moved.pdb"
    .type = str
  alignment_style = *local global
```

```
    .type = choice
  gap_opening_penalty = 20
    .type = float
  gap_extension_penalty = 2
    .type = float
  similarity_matrix = *blosum50 dayhoff
    .type = choice
}
""")
```

`master_params` is a Phil `scope` instance with a `super` sub-scope (for clarity) that defines the parameters we need. `fixed` and `moving` are input files names, `moved` is an output file name. The other parameters are for `mmtbx.alignment.align`.

All parameters can be modified from the command line, e.g.:

```
mmtbx.super fixed=first.pdb moving=second.pdb similarity_matrix=dayhoff
```

This is enabled with the following code fragments in `super.py`:

```
import libtbx.phil.command_line
phil_objects = []
argument_interpreter = libtbx.phil.command_line.argument_interpreter(
  master_params=master_params, home_scope="super")

...

    try: command_line_params = argument_interpreter.process(arg=arg)
    except: raise Sorry("Unknown file or keyword: %s" % arg)
    else: phil_objects.append(command_line_params)
```

As an added convenience, bare file names are also recognized, e.g. this is an alternative to the command above:

```
mmtbx.super first.pdb second.pdb similarity_matrix=dayhoff
```

The complete code (including the fragments above) for supporting this generality is:

```
def run(args):
  phil_objects = []
  argument_interpreter = libtbx.phil.command_line.argument_interpreter(
    master_params=master_params, home_scope="super")
  fixed_pdb_file_name = None
  moving_pdb_file_name = None
  for arg in args:
    if (os.path.isfile(arg)):
      if (fixed_pdb_file_name is None): fixed_pdb_file_name = arg
      elif (moving_pdb_file_name is None): moving_pdb_file_name = arg
      else: raise Sorry("Too many file names.")
    else:
      try: command_line_params = argument_interpreter.process(arg=arg)
      except: raise Sorry("Unknown file or keyword: %s" % arg)
      else: phil_objects.append(command_line_params)
```

At this point we have to consolidate the two possible sources of information: bare file names (stored under `fixed_pdb_file_name` and `moving_pdb_file_name`) and assignments via `fixed=...` or `moving=...`. First, we combine all the Phil assignments (stored under `phil_objects`) into one `working_params` object and use the `extract()` method to get easy access to the definitions (see Newsletter No. 5):

```
working_params = master_params.fetch(sources=phil_objects)
params = working_params.extract()
```

Now we override the Phil assignments with the bare file names if available, or generate an error message if a file name is missing:

```
if (fixed_pdb_file_name is None):
  if (params.super.fixed is None): raise_missing("fixed")
else:
  params.super.fixed = fixed_pdb_file_name
if (moving_pdb_file_name is None):
  if (params.super.moving is None): raise_missing("moving")
else:
  params.super.moving = moving_pdb_file_name
```

`raise_missing()` is a simple function raising an informative exception (see `super.py`), e.g.:

```
Sorry: Missing file name for moving structure:
  Please add
    moving=file_name
  to the command line to specify the moving structure.
```

5.3 Processing of PDB input files

With all the input parameters consolidated in the `params` object above, reading in the PDB files is simple:

```
fixed_pdb = iotbx.pdb.input(file_name=params.super.fixed)
moving_pdb = iotbx.pdb.input(file_name=params.super.moving)
```

For both files we have to extract the sequence of residue names and corresponding C-alpha coordinates. This is implemented as a function that we call twice:

```
fixed_seq, fixed_sites, fixed_site_flags = extract_sequence_and_sites(
  pdb_input=fixed_pdb)
moving_seq, moving_sites, moving_site_flags = extract_sequence_and_sites(
  pdb_input=moving_pdb)
```

For the complete implementation of `extract_sequence_and_sites()` please refer to `super.py`. For simplicity, the function only considers the first `MODEL` in the pdb file, and for each chain only the first conformer (as derived from the `altloc` symbols):

```
model = pdb_input.construct_hierarchy().models()[0]
for chain in model.chains():
  selected_residues = chain.conformers()[0].residue_class_selection(
    class_name="common_amino_acid")
  residues = chain.conformers()[0].residues()
  for ires in selected_residues:
    ...
```

Another simplification is the selection of `"common_amino_acid"` residues only. For these, the residue names are translated to one-letter codes which are collected in a `seq` list:

```
import mmtbx.amino_acid_codes
...
  seq = []
  ...
      resi = residues[ires]
      resn = resi.name[0:3]
      single = mmtbx.amino_acid_codes.one_letter_given_three_letter[resn]
      seq.append(single)
```

The rest of the body of the loop over the selected residues extracts the C-alpha coordinates if available:

```
from cctbx.array_family import flex
...
  sites = flex.vec3_double()
  use_sites = flex.bool()
  ...
      use = False
      xyz = (0,0,0)
      for atom in resi.atoms():
        if (atom.name == " CA "):
          xyz = atom.xyz
          use = True
          break
      sites.append(xyz)
      use_sites.append(use)
```

The coordinates are stored under `sites`. A corresponding `use_sites` array of bools (False or True) stores if a C-alpha atom was found or not. Finally the collected sequence, coordinates and use flags are returned with:

```
return "".join(seq), sites, use_sites
```

The list of one-letter codes is converted to a plain string on the fly. The plain string is more convenient to work with in the following steps.

5.4 Sequence alignment

Sequence alignment is now a simple call of `mmtbx.alignment.align` as discussed before. The function call parameters are taken directly from the Phil `params` object:

```
align_obj = mmtbx.alignment.align(
  seq_a=fixed_seq,
  seq_b=moving_seq,
  gap_opening_penalty=params.super.gap_opening_penalty,
  gap_extension_penalty=params.super.gap_extension_penalty,
  similarity_function=params.super.similarity_matrix,
  style=params.super.alignment_style)
```

From the `align_obj` we extract the `alignment` as shown before, but we spend a little more effort to produce nice output:

```
alignment = align_obj.extract_alignment()
matches = alignment.matches()
equal = matches.count("|")
similar = matches.count("*")
total = len(alignment.a) - alignment.a.count("-")
alignment.pretty_print(
  matches=matches,
  block_size=50,
  n_block=1,
  top_name="fixed",
  bottom_name="moving",
  comment="""... see super.py ... """)
```

This code produces, e.g.:

```
The alignment used in the superposition is shown below.

The sequence identity (fraction of | symbols) is 55.1%
of the aligned length of the fixed molecule sequence.
```

```
The sequence similarity (fraction of | and * symbols) is 75.5%
of the aligned length of the fixed molecule sequence.

              12345678901234567890123456789012345678901234567890

fixed         VTDNIMKHSKNPIIIVVSNPLDIMTHVAWVRSGLPKERVIGMAGVLDAA
              *  ||*||| * ||*|||||*|**|*|||  ||||  |*||    ||*|
moving        IIPNIVKHSPDCIILVVSNPVDVLTYVAWKLSGLPMHRIIGSGCNLDSA
```

5.5 Least-squares superposition

To keep this example simple, in the least-squares superposition we want to use only the C-alpha coordinates of matching residues, i.e. residues with a `|` or `*` symbol in the output above. This information is stored under `matches` as obtained above. We also have to check if the C-alpha coordinates are available for both of the matching residues. This information is stored under `fixed_site_flags` and `moving_site_flags`. The matching + available C-alpha coordinates are obtained with this code:

```
fixed_sites_sel = flex.vec3_double()
moving_sites_sel = flex.vec3_double()
for ia,ib,m in zip(alignment.i_seqs_a, alignment.i_seqs_b, matches):
  if (m not in ["|", "*"]): continue
  if (fixed_site_flags[ia] and moving_site_flags[ib]):
    fixed_sites_sel.append(fixed_sites[ia])
    moving_sites_sel.append(moving_sites[ib])
```

Computing the superposition and printing out the RMSD between the aligned, superposed C-alpha atoms is now very simple:

```
lsq_fit = superpose.least_squares_fit(
  reference_sites=fixed_sites_sel,
  other_sites=moving_sites_sel)
rmsd = fixed_sites_sel.rms_difference(lsq_fit.other_sites_best_fit())
print "  RSMD between the aligned C-alpha atoms: %.3f" % rmsd
```

5.6 Export of moved coordinates

As the final step, `mmtbx.super` applies the rotation and translation obtained in the least squares fit to all original coordinates in `moving_pdb` and writes out the modified atom records:

```
print "Writing moved pdb to file: %s" % params.super.moved
out = open(params.super.moved, "w")
for serial, label, atom in zip(moving_pdb.atom_serial_number_strings(),
                               moving_pdb.input_atom_labels_list(),
                               moving_pdb.atoms()):
  print >> out, iotbx.pdb.format_atom_record(
    record_name={False: "ATOM", True: "HETATM"}[atom.hetero],
    serial=int(serial),
    name=label.name(),
    altLoc=label.altloc(),
    resName=label.resname(),
    resSeq=label.resseq,
    chainID=label.chain(),
    iCode=label.icode(),
    site=lsq_fit.r * matrix.col(atom.xyz) + lsq_fit.t,
    occupancy=atom.occ,
    tempFactor=atom.b,
    segID=atom.segid,
    element=atom.element,
    charge=atom.charge)
```

All of the information on the input ATOM or HETATM records is passed through as-is, except for the coordinates. We make use of the `scitbx.matrix` facilities again to apply `lsq_fit.r` and `lsq_fit.t` to `atom.xyz`. `format_atom_record()` is a very simple function, essentially just a Python string formatting statement which could also be spelled out inline. However, the assignment of the data items to function parameters is easier to read and understand than the raw formatting statement, and the function handles some subtleties (e.g. overflowing serial and resSeq) that are easily overlooked. Using a central function ensures that subtle and rare problems like this are fixed everywhere once they are discovered.

6 Overview of refinement development

In the crystallographic context structure refinement means optimization of certain target functions by modifying various model parameters. Depending on several factors (e.g. available data, model quality and size) the model can be parameterized in different ways, as grouped or individual atomic parameters. Individual atomic parameters are coordinates, isotropic or anisotropic ADPs (atomic displacement parameters), and occupancy factors. Grouped parameterizations are rigid body, group ADP, TLS, group occupancy, and overall anisotropic scale factor. All of these except group occupancy refinement are currently implemented in `phenix.refine`.

The most recent version of `phenix.refine` allows automatic refinement of any combination of parameters for any part or combination of parts of the model. To the best of our knowledge this is a unique feature among existing crystallographic software.

To give an example, for a molecule with three chains A, B, and C, the command:

```
phenix.refine model.pdb data.mtz \
  strategy=rigid_body+individual_sites+individual_adp+tls \
  sites.rigid_body="chain A" \
  sites.individual="chain B" \
  adp.tls="chain A" \
  adp.tls="chain C"
```

will perform refinement of:

- chain A as a rigid body
- individual isotropic ADPs for the whole molecule
- individual coordinates for chain B
- TLS parameters for chain A and chain C

More information about `phenix.refine` is available at http://phenix-online.org/download/cci_apps/ .

In the following we will highlight a few selected open-source modules supporting `phenix.refine`.

6.1 Rigid body refinement

The core machinery for rigid body refinement is located in `cctbx_sources/mmtbx/mmtbx/refinement/rigid_body.py`. A typical call is:

```
rigid_body_manager = mmtbx.refinement.rigid_body.manager(
  fmodel                  = fmodel,
  selections              = rigid_body_selections,
  refine_r                = True,
  refine_t                = True,
  convergence_test        = True,
  nref_min                = 1000,
```

```
  max_iterations          = 25,
  use_only_low_resolution = False,
  high_resolution         = 2.0,
  low_high_res_limit      = 6.0,
  max_low_high_res_limit  = 8.0,
  bulk_solvent_and_scale  = True,
  bss                     = bulk_solvent_and_scale_parameters,
  euler_angle_convention  = "xyz",
  log                     = log)
rotations    = rigid_body_manager.total_rotation,
translations = rigid_body_manager.total_translation
```

This performs L-BFGS minimization of a crystallographic target w.r.t. `6*len(rigid_body_selections)` parameters. The model (`xray_structure`), crystallographic data (Fobs, ...), and target definition are held by `fmodel`. The selections can cover either the whole molecule or selected parts. Atoms that are not selected are fixed during refinement. Depending on the parameters, it can perform either conventional rigid body refinement in a selected resolution range or use more sophisticated multi-zone protocol where the refinement starts in low resolution zone (defined by `nref_min`) and proceeds with the whole set of reflections. Bulk solvent parameters and scale parameters are updated automatically if the model is shifted more than a certain threshold.

6.2 Grouped isotropic ADP refinement

The code for grouped isotropic ADP refinement resides in `cctbx_sources/mmtbx/mmtbx/refinement/group_b.py`. A typical call is:

```
mmtbx.refinement.group_b.manager(
  fmodel                      = fmodel,
  selections                  = group_adp_selections,
  convergence_test            = True,
  max_number_of_iterations    = 25,
  number_of_macro_cycles      = 3,
  run_finite_differences_test = False,
  log                         = log)
```

This performs refinement of one isotropic ADP per selected group. Non-specific input parameters are similar to those in rigid body refinement module. The refinable parameters are a shift in isotropic ADP for each group, which are added to the original ADPs. The group-specific shifts are applied to both isotropic and anisotropic atoms. For the latter the shift is added to the three diagonal elements of the ADP tensor. In particular this is important for TLS refinement where the atoms in TLS groups are anisotropic and the group ADP refinement is used. ADPs of non-selected atoms are unchanged.

6.3 TLS refinement

This is the most complex code mentioned here and is located in the directories `cctbx_sources/mmtbx/mmtbx/tls` and `cctbx_sources/mmtbx/tls` (Python and C++ code, respectively). The file `cctbx_sources/mmtbx/mmtbx/tls/tools.py` contains the class:

```
class tls_refinement(object):
   def __init__(self, fmodel,
                      model,
                      selections,
                      refine_T,
                      refine_L,
                      refine_S,
                      number_of_macro_cycles,
                      max_number_of_iterations,
                      start_tls_value = None,
                      run_finite_differences_test = False,
```

```
                        eps = 1.e-6,
                        out = None,
                        macro_cycle = None):
```

which performs all principal steps in its constructor, including extraction of start TLS parameters from the current ADPs (extracted from `fmodel.xray_structure`) and the current TLS parameters (zero in the first macro-cycle), L-BFGS minimization of a crystallographic target w.r.t. the TLS parameters, split of total ADPs into local and TLS components, and enforcement of positive-definiteness of the final ADP tensors for each individual atom. These operations are exposed as helper functions in `tools.py` which can also be called individually.

6.4 Individual coordinates, ADP and occupancies

The file `cctbx_sources/mmtbx/mmtbx/refinement/minimization.py` is one of the most matrue files in the mmtbx and is the main driver for restrained refinement of individual coordinates, ADPs (isotropic or anisotropic) and occupancies for selected atoms using X-ray and/or neutron data. E.g. to perform coordinate refinement:

```
mmtbx.refinement.minimization.lbfgs(
  restraints_manager       = restraints_manager,
  fmodel                   = fmodel,
  model                    = model,
  refine_xyz               = True,
  lbfgs_termination_params = lbfgs_termination_params,
  wx                       = xray_term_weight,
  wc                       = geometry_term_weight,
  verbose                  = 0)
```

The `model` object contains selection information to determine which atoms are refined and which are fixed.

7 Acknowledgments

We gratefully acknowledge the financial support of NIH/NIGMS under grant number P01GM063210. Our work was supported in part by the US Department of Energy under Contract No. DE-AC02-05CH11231.

8 References

Dayhoff, M.O. (1978). Atlas of Protein Sequence and Structure, Vol. 5 suppl. 3, 345-352.
Gotoh, O. (1982). J. Mol. Biol. 162, 705-708.
Grosse-Kunstleve, R.W., Adams, P.D. (2003). Newsletter of the IUCr Commission on Crystallographic Computing, 1, 28-38.
Grosse-Kunstleve, R.W., Afonine, P.V., Adams, P.D. (2004). Newsletter of the IUCr Commission on Crystallographic Computing, 4, 19-36.
Grosse-Kunstleve, R.W., Afonine, P.V., Sauter, N.K., Adams, P.D. (2005). Newsletter of the IUCr Commission on Crystallographic Computing, 5, 69-91.
Henikoff & Henikoff (1992). PNAS 89, 10915-10919
Kabsch, W. (1976). Acta Cryst. A32, 922-923.
Kabsch, W. (1978). Acta Cryst. A34, 827-828.
Kearsley, S.K. (1989). Acta Cryst. A45, 208-210.
Needleman, S. & Wunsch, C. (1970). J. Mol. Biol. 48(3), 443-53.
Smith, T.F. & Waterman M.S. (1981). J. Mol. Biol. 147, 195-197.